

Generative Music with Haskell

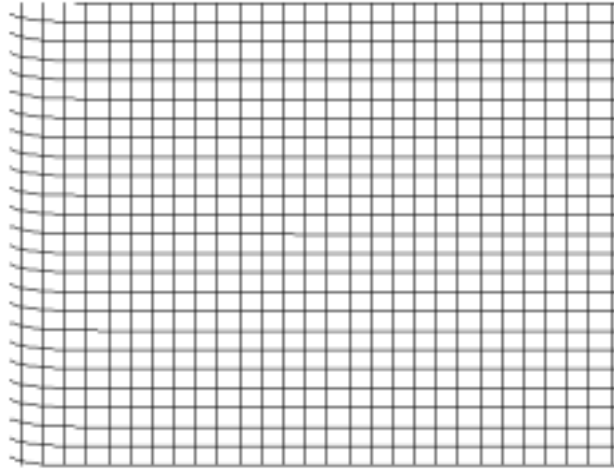
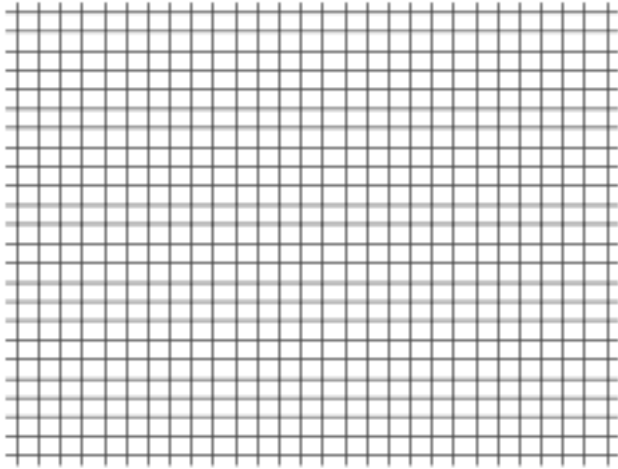
Martin Valdes de Leon



Physics

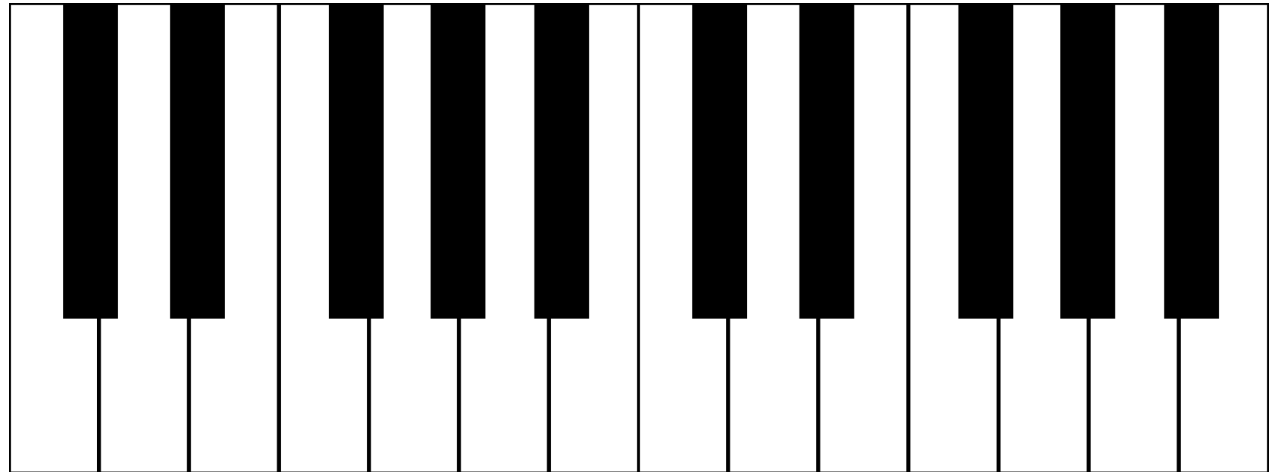
What is sound?

- A vibration that propagates as a compressive/expansive wave through a transmission medium
- A vibration that propagates as a pressure wave through air



What characterizes sound?

- Loudness
- Pitch
 - Does all sound have pitch?
 - Are all pitches audible by humans?
 - 20 Hz to 20 kHz
 - Chromatic scale

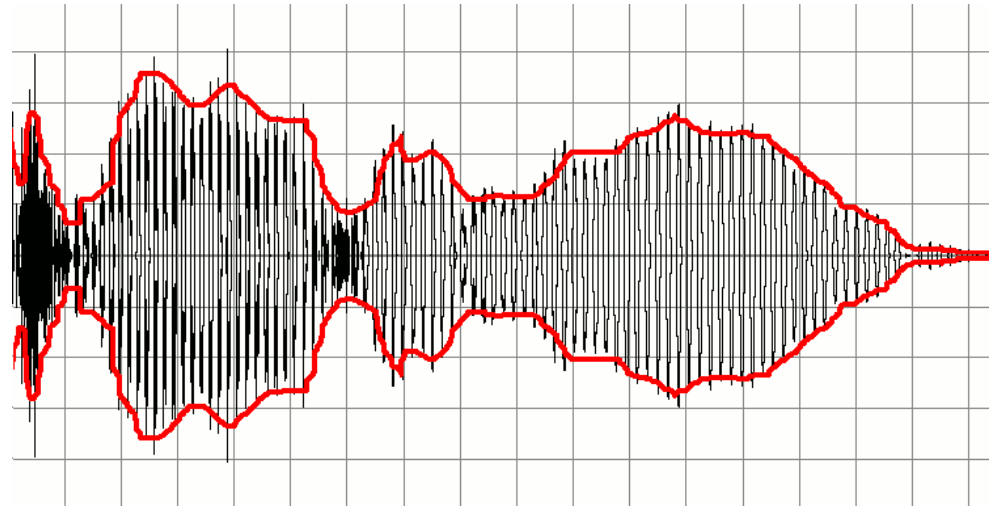
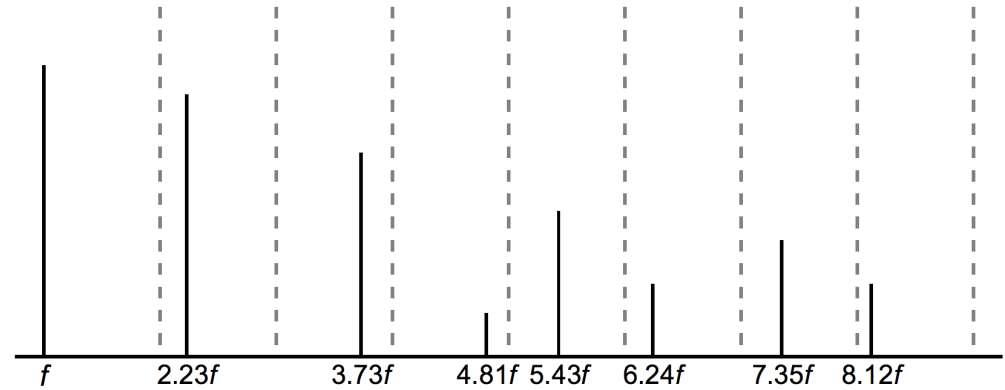


What characterizes sound?

- Is that it?
- Why does e.g. a flute sound different than a piano, when playing the same note at a similar loudness level?

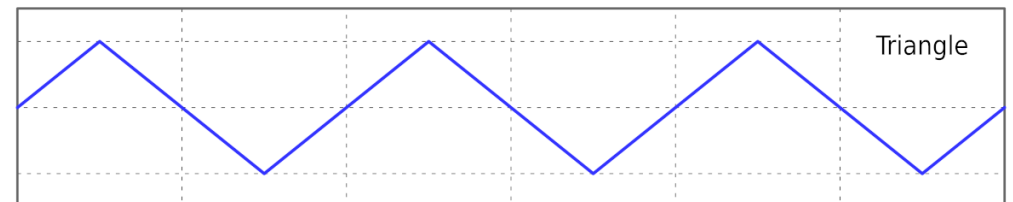
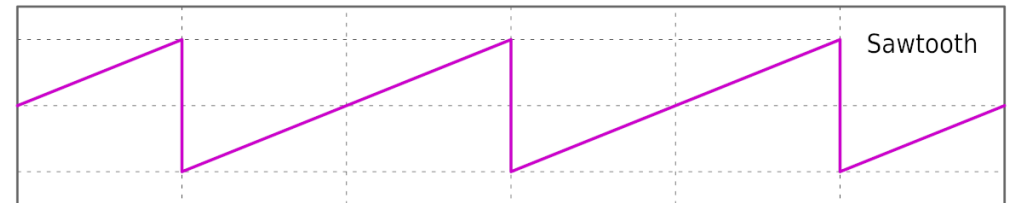
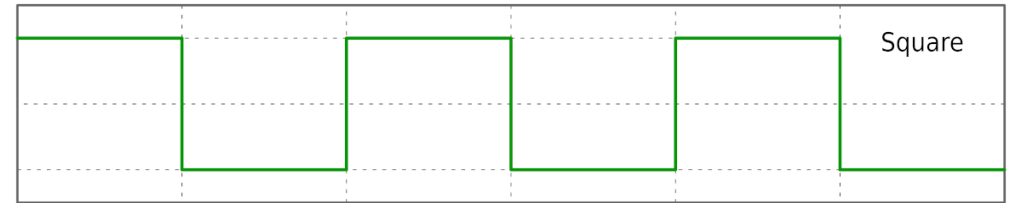
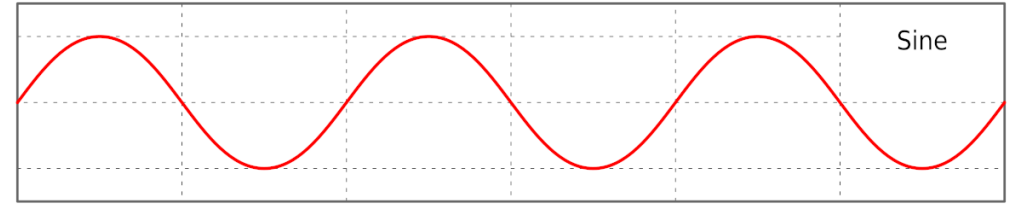
What characterizes sound?

- Timbre /Color
 - Partial
 - Envelopes



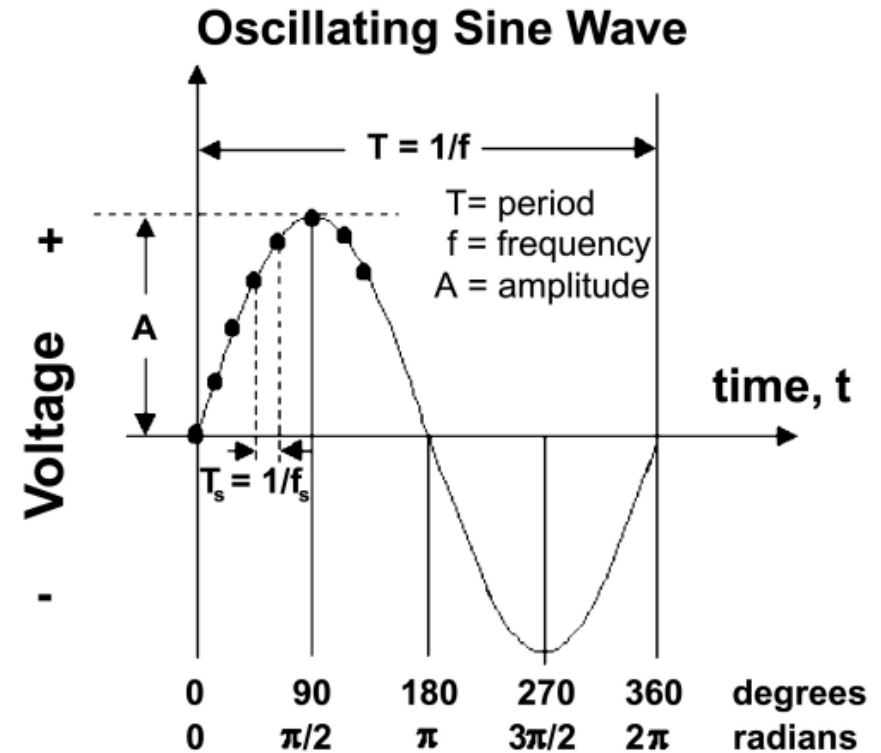
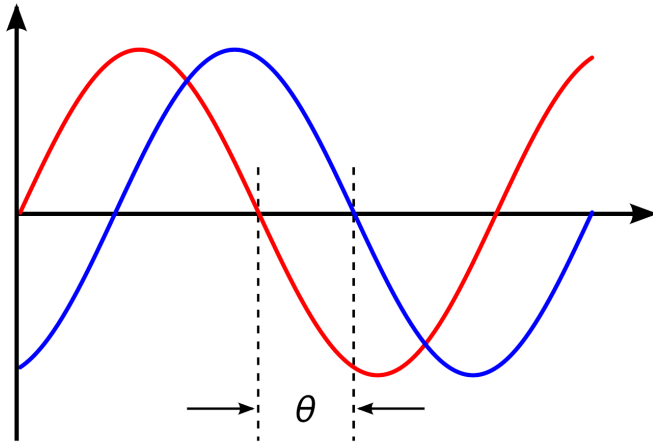
Simple waveforms

- Sine wave
- Square wave
 - Pulse width / duty cycle
- Sawtooth wave
- Triangle wave



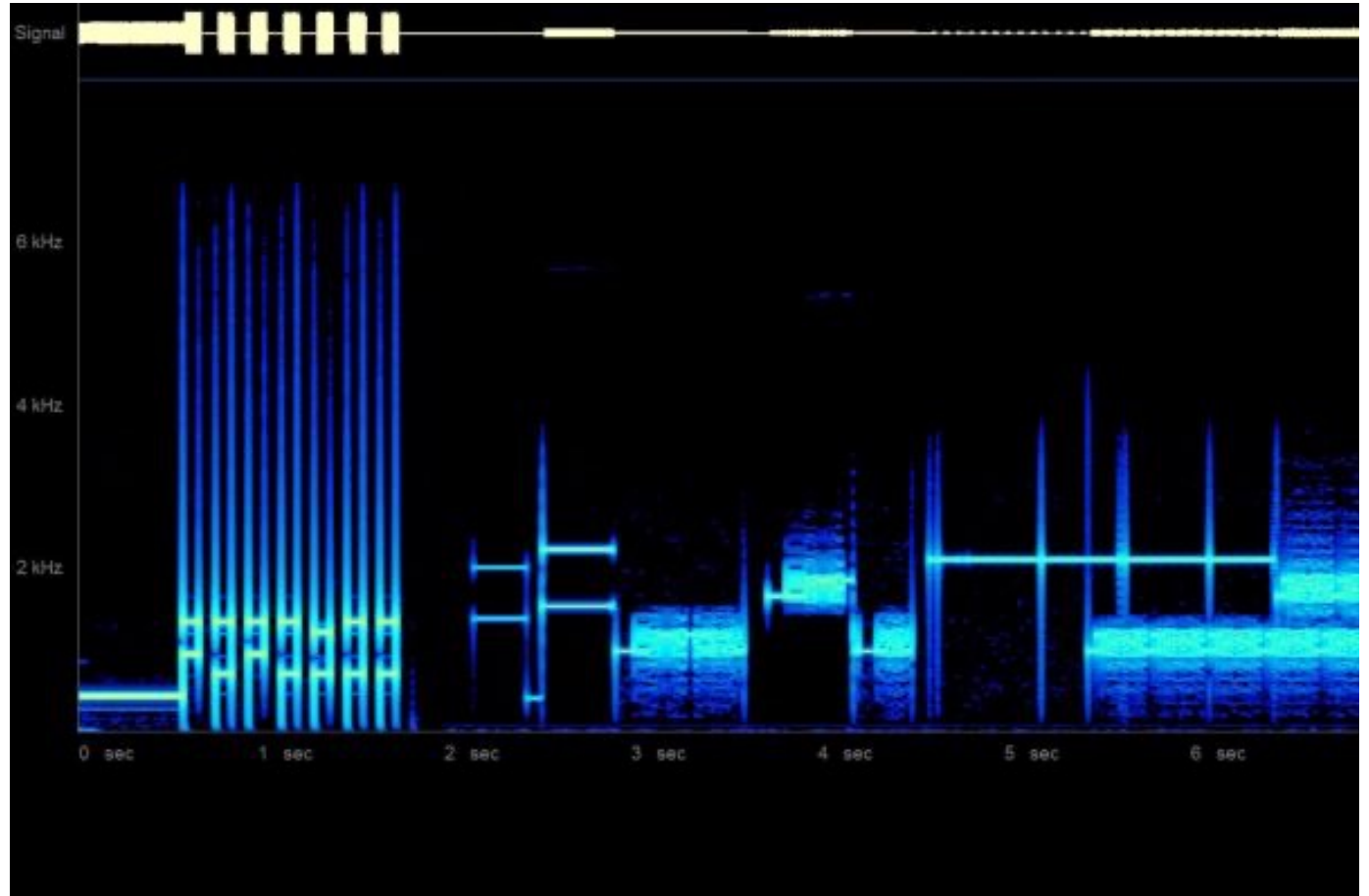
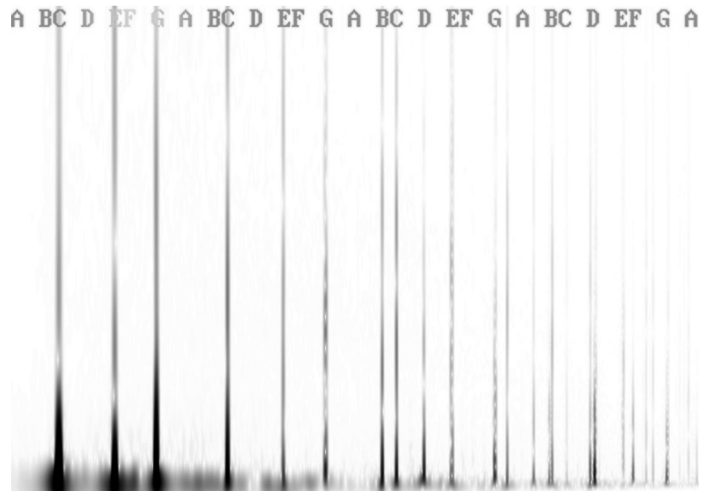
Simple waveforms

- Properties
 - Amplitude
 - Frequency
 - Phase



Time vs Frequency domains

- Amplitud vs Time
- Amplitude vs Frequency
- Spectrogram



Time vs Frequency domains

- Fourier transform
 - Any function can be represented as a combination of a potentially infinite number of trigonometric functions

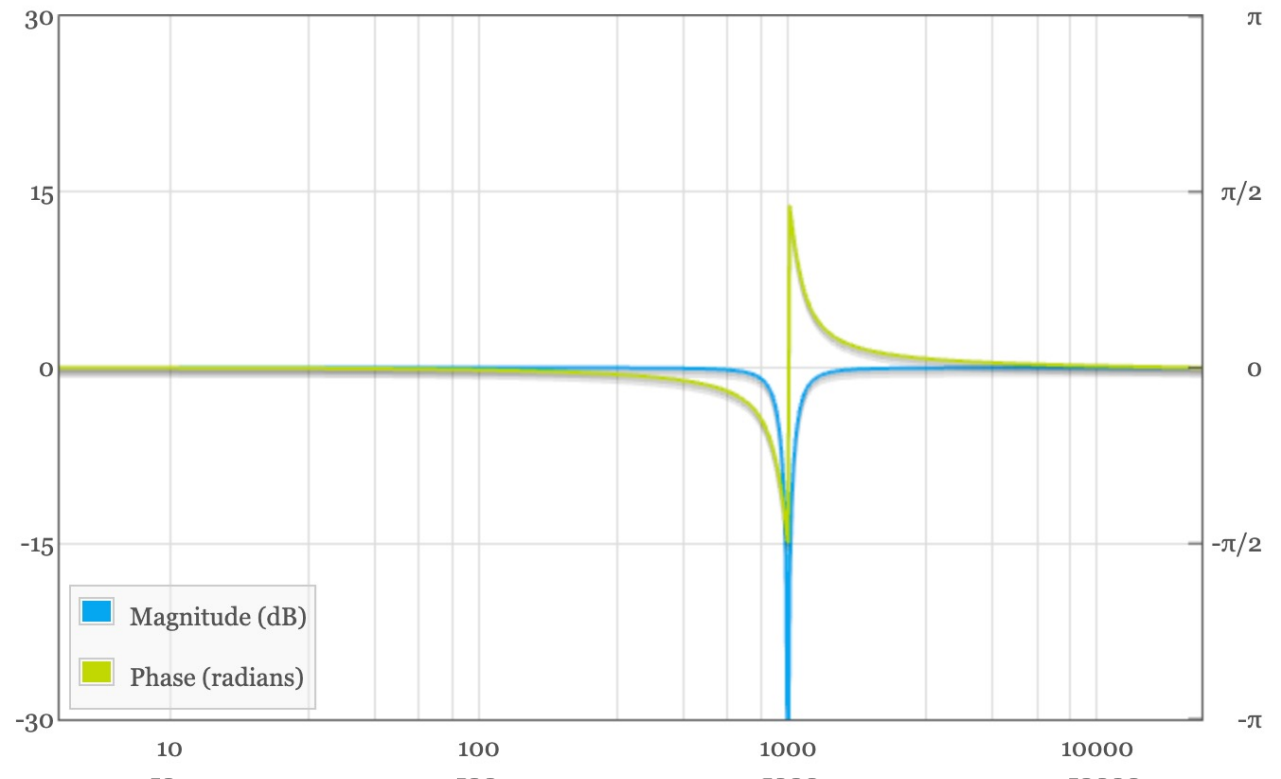


Time vs Frequency domains

- Partial / harmonics
 - Square wave: Odd harmonics only, $A = 1/n$
 - Sawtooth wave: Odd and even harmonics, $A = (-1)^n/n$
- **Additive synthesis**

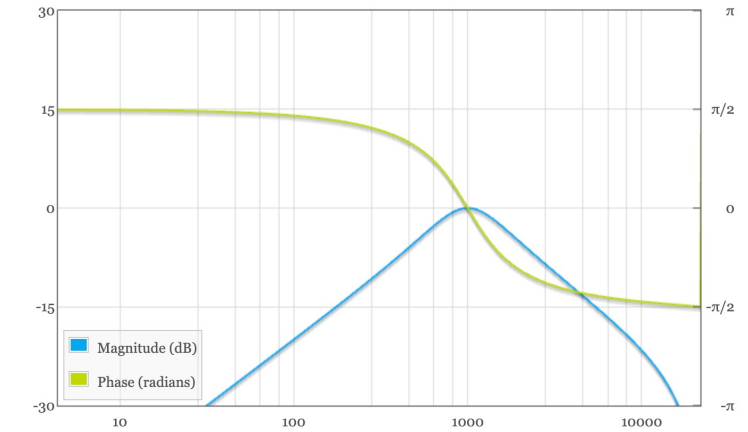
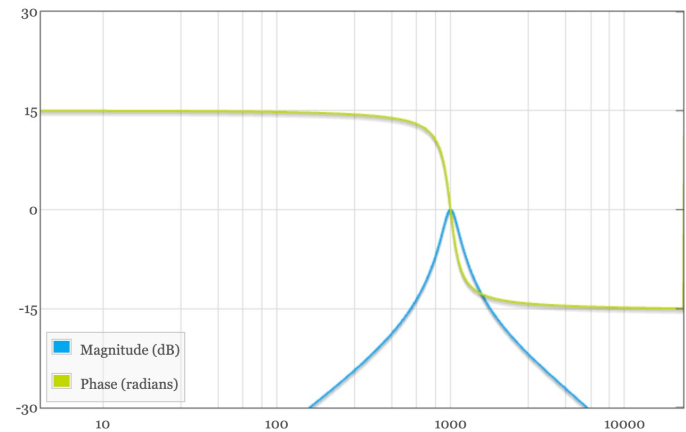
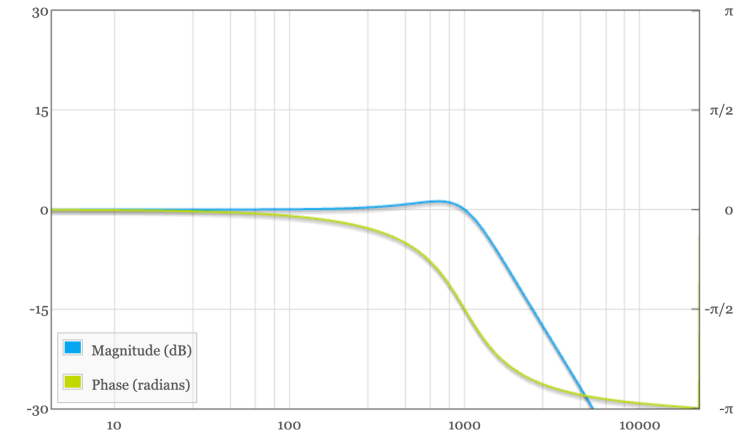
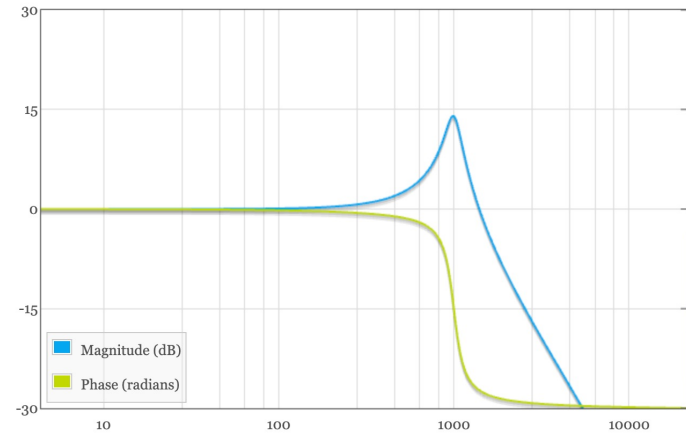
Filtering

- Operate on the frequency domain, either attenuating or amplifying certain frequencies
- Types of filters
 - Low-pass
 - High-pass
 - Band-pass
 - Notch / Band-stop



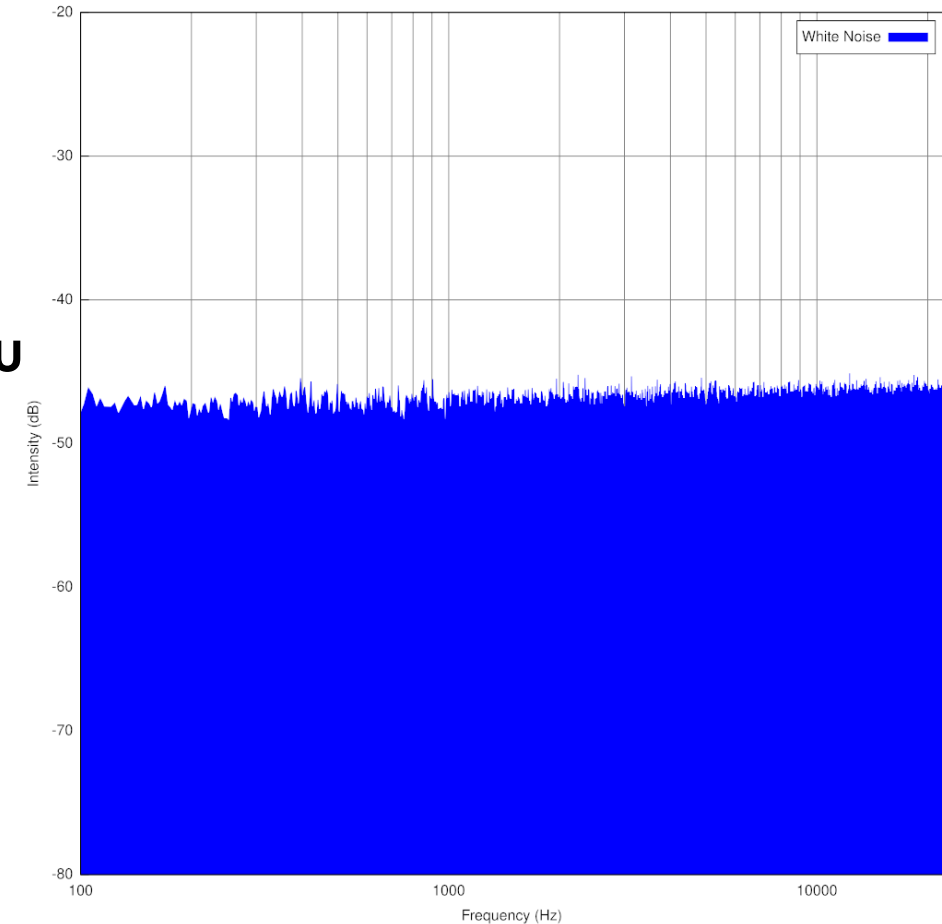
Filtering

- Characteristics of filters
 - Cutoff frequency
 - Q
 - Bandwidth
 - Gain/Attenuation
- Subtractive synthesis



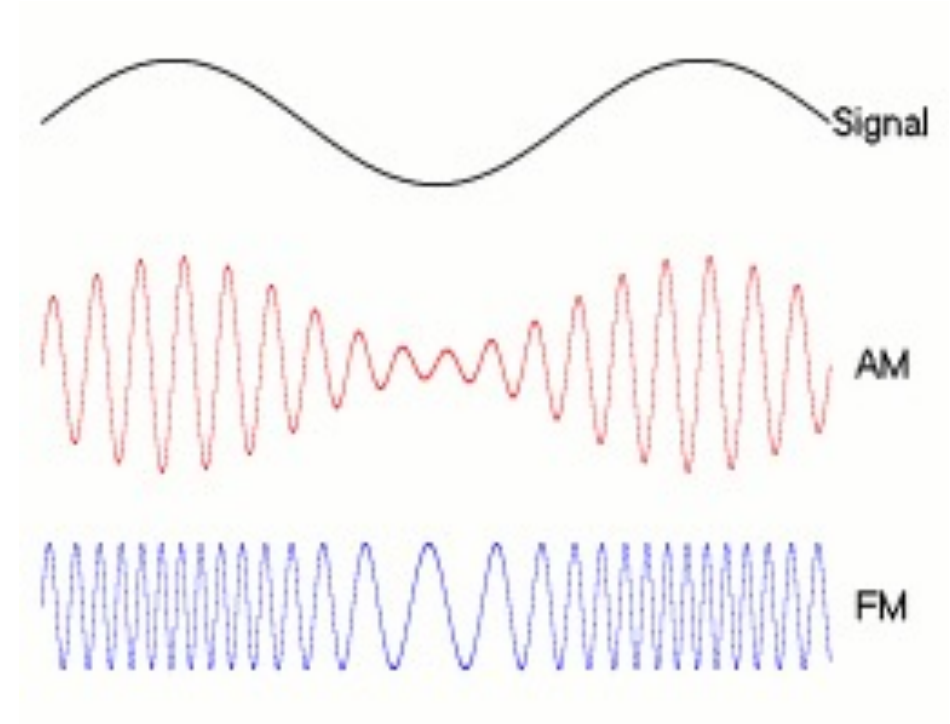
Noise

- White noise
 - Uniformly distributed random values
 - Contains all frequencies with equal amplitude
 - Used as a source for subtractive synthesis
 - Used as a source of randomness
 - Sample-and-hold



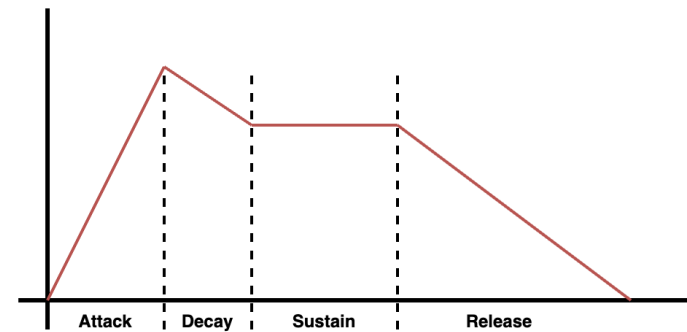
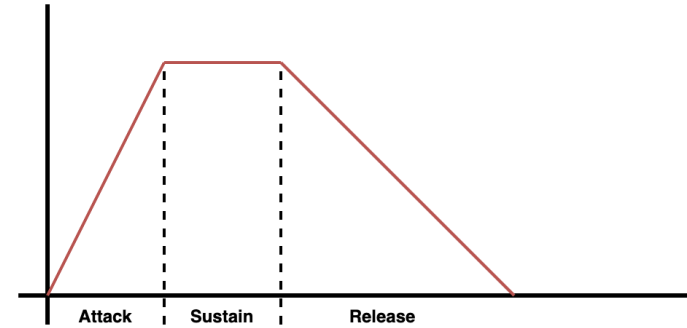
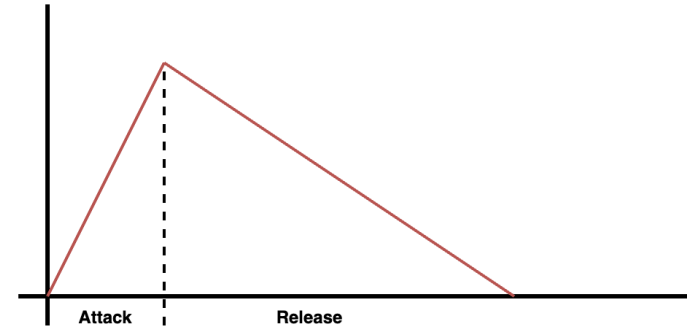
Modulation

- Sources
 - Waveforms
 - LFO
 - Envelopes
 - Random?
- Targets
 - Anything!
 - Pitch
 - Amplitude
 - Filter cutoff frequency



Envelopes

- Common envelopes
 - AR
 - ASR
 - ADSR



Haskell

Euterpea2

- Digital signal processing
 - Generating sound from scratch
- MIDI-based composition

Euterpea2

- **Render WAV files**
- **Live MIDI playback**

Arrow Basics



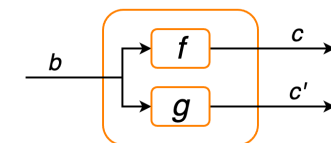
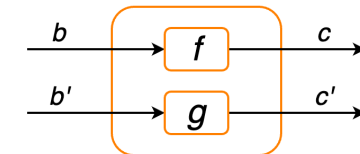
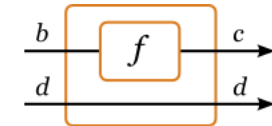
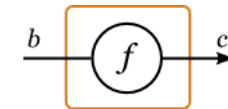
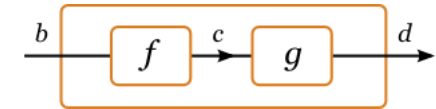
- General abstraction over computation
- Euterpea2 uses arrows to model signal processing
 - We want to generate and/or process audio samples at a suitable rate
 - Nyquist–Shannon
 - 44.1 kHz

Arrow Basics

```
class Category cat where
  -- | the identity morphism
  id :: cat a a
  -- | morphism composition
  (.) :: cat b c -> cat a b -> cat a c

class Category a => Arrow a where
  -- | Lift a function to an arrow.
  arr :: (b -> c) -> a b c
  -- | Send the first component of the input through the argument
  --   arrow, and copy the rest unchanged to the output.
  first :: a b c -> a (b,d) (c,d)
  --   The default definition may be overridden with a more efficient
  --   version if desired.
  second :: a b c -> a (d,b) (d,c)
  -- | Split the input between the two argument arrows and combine
  --   their output. Note that this is in general not a functor.
  (***) :: a b c -> a b' c' -> a (b,b') (c,c')
  -- | Fanout: send the input to both argument arrows and combine
  --   their output.
  (&&&) :: a b c -> a b c' -> a b (c,c')

returnA :: Arrow a => a b b
returnA = id
-- or (arr id), when id :: a -> a
```



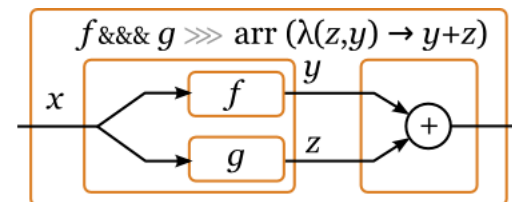
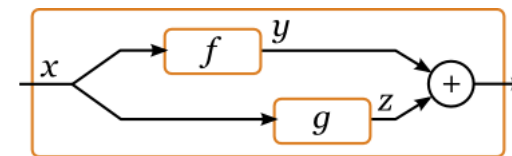
Arrow Basics

```
-- do-notation / Arrow syntax
addA :: Arrow a => a b Int -> a b Int -> a b Int
addA f g = proc x -> do
    y <- f -< x
    z <- g -< x
    returnA -< y + z
```

```
-- Or we could try and be "smart"...
addA f g = f &&& g >>> arr (\(y, z) -> y + z)
```

```
-- Constant Arrow
constA :: (Arrow arr) => b -> arr a b
constA = arr . const
```

```
-- foldA
foldA :: (Foldable t, Arrow arr) => (b -> c -> c) -> c -> t (arr a b) ->
arr a c
foldA f c = foldr (\sfb sfc -> sfb &&& sfc >>> arr (uncurry f)) (constA c)
```

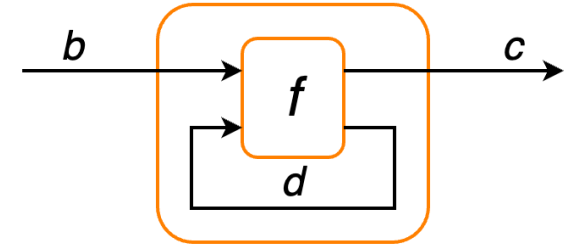


Arrow Basics

```
class Arrow a => ArrowLoop a where  
  loop :: a (b,d) (c,d) -> a b c
```

```
class ArrowLoop a => ArrowCircuit a where  
  -- / A delay component  
  delay :: b -> a b b
```

```
counter :: ArrowCircuit a => a Bool Int  
counter = proc reset -> do  
  rec output <- returnA -< if reset then 0 else next  
  next <- delay 0 -< output + 1  
  returnA -< output
```



Euterpea2 Types

```
-- An arrow `a b c`, with a phantom type `p`
-- Has Category & Arrow instances if a does
newtype ArrowP a p b c = ArrowP { strip :: a b c }

-- A way of generating `b`s from `a`s.
-- Has Category & Arrow instances
newtype SF a b = SF { runSF :: (a -> (b, SF a b)) }

-- A way of generating `b`s from `a`s,
-- Has Category & Arrow instances, with an extra parameter `clk`
type Signal clk a b = ArrowP SF clk a b
type SigFun clk a b = ArrowP SF clk a b

-- A way of passing the sample rate at the type level
class Clock p where
    rate :: p -> Double

-- No data constructors!
data AudRate
data CtrRate

instance Clock AudRate where
    rate _ = 44100

instance Clock CtrRate where
    rate _ = 4410

-- A way of generating `b`s from `a`s, at a specific rate
type AudSF a b = SigFun AudRate a b
type CtrSF a b = SigFun CtrRate a b
```


Wavetable Synthesis

```
data Table = Table
    !Int          -- size
    !(UArray Int Double) -- table implementation
    !Bool         -- Whether the table is normalized

type TableSize = Int
type PartialStrenght = Double

tableLinear :: TableSize -> StartPt -> [(SegLength, EndPt)] -> Table
tableLinearN :: TableSize -> StartPt -> [(SegLength, EndPt)] -> Table
tableSines :: TableSize -> [PartialStrenght] -> Table
tableSinesN :: TableSize -> [PartialStrenght] -> Table

-- e.g.
-- oscFixed frequency :: AudSF () Double
oscFixed :: forall p a . (Clock p, ArrowCircuit a) => Double -> ArrowP a p () Double
-- osc wavetable phase :: AudSF Double Double
-- But what about the frequency?
osc :: (Clock p, ArrowCircuit a) => Table -> Double -> ArrowP a p Double Double
```

Envelopes

```
-- [y0, y1, y2, ...] => (0, y0) (x1, y1) (x2, y2) ...
--   [ x1, x2, ...]

-- IMPORTANT: This function holds the last SLOPE, not the last VALUE
envLineSeg :: Clock p => [Double] -> [Double] -> Signal p () Double

-- IMPORTANT: This function holds the last RATE, not the last VALUE
envExponSeg :: Clock p => [Double] -> [Double] -> Signal p () Double
```

Effects

```
-- Fixed delay
delayLine :: forall p . Clock p => Double -> Signal p Double Double

-- Dynamic delay, up to a fixed maximum
-- (signal, delay)
delayLine1 :: forall p . Clock p => Double -> Signal p (Double, Double) Double
```

Filters

```
-- (signal, frequency)
filterLowPass :: forall p . Clock p => Signal p (Double, Double) Double
filterHighPass :: Clock p => Signal p (Double, Double) Double

-- (signal, frequency)
filterLowPassBW :: forall p . Clock p => Signal p (Double, Double) Double
filterHighPassBW :: forall p . Clock p => Signal p (Double, Double) Double
-- (signal, frequency, bandwidth)
filterBandPassBW :: forall p . Clock p => Signal p (Double, Double, Double) Double
filterBandStopBW :: forall p . Clock p => Signal p (Double, Double, Double) Double

-- loop time in seconds
-- (signal, reverb time)
filterComb :: Clock p => Double -> Signal p (Double, Double) Double
```

White Noise

```
-- seed for the RNG  
noiseWhite :: Int -> Signal p () Double
```

Rendering to WAV

```
-- Double and (Double, Double) are instances of AudioSample
-- e.g., mono or stereo signals
outFile :: forall a p. (AudioSample a, Clock p) =>
  String -- ^ Filename to write to.
  -> Double -- ^ Duration of the wav in seconds.
  -> Signal p () a -- ^ Signal representing the sound.
  -> IO ()
```

And now?

- Clone the workshop repository
 - <https://github.com/mvaldesdeleon/generative-music-with-haskell>
- Get your environment up and running
- Try to implement the different challenges
- Play around on your own

Testing

- Visually inspect the waveform
- Visually inspect the spectrogram
- Listen (carefully!)

Resources

- Arrows
 - <https://www.haskell.org/arrows/syntax.html>
 - <https://hackage.haskell.org/package/base-4.20.0.1/docs/Control-Arrow.html>
 - https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/arrows.html
- Euterpea2
 - <https://www.euterpea.com/>
 - <https://hackage.haskell.org/package/Euterpea-2.0.7>
 - https://www.euterpea.com/wp-content/uploads/2016/12/euterpea_signal_quick_reference.pdf
 - https://www.euterpea.com/wp-content/uploads/2016/12/Euterpea_Quick_Reference.pdf
- Audacity
 - <https://www.audacityteam.org/>

Now go make some noise!

Please don't fuck up your ears...

Extras

Pitches

```
type AbsPitch = Int -- MIDI, 0 to 127, ±1 = ±1 semitone, 60 = C4

data PitchClass = Cff | Cf | C | Dff | Cs | Df | Css | D | Eff | Ds
                | Ef | Fff | Dss | E | Ff | Es | F | Gff | Ess | Fs
                | Gf | Fss | G | Aff | Gs | Af | Gss | A | Bff | As
                | Bf | Ass | B | Bs | Bss
    deriving (Show, Eq, Ord, Read, Enum, Bounded)
type Octave = Int
type Pitch = (PitchClass, Octave)
```

Notes

```
type Dur = Rational
bn, wn, hn, qn, en, sn, tn, sfm :: Dur

data Primitive a = Note Dur a | Rest Dur
  deriving (Show, Eq, Ord)

data Music a = Prim (Primitive a)
  | Music a :+: Music a -- sequential composition
  | Music a :=: Music a -- parallel composition
  | Modify Control (Music a) -- modifier (e.g., instrument selection)
  deriving (Show, Eq, Ord)

line, chord :: [Music a] -> Music a
line = foldr (:+:) (rest 0)
chord = foldr (:=:) (rest 0)

note :: Dur -> a -> Music a
note d p = Prim (Note d p)

rest :: Dur -> Music a
rest d = Prim (Rest d)
```

Rendering to SF

```
type Mono p = Signal p () Double
type Stereo p = Signal p () (Double, Double)
type Volume = Int

type Instr a = Dur -> AbsPitch -> Volume -> [Double] -> a
-- Instr (Mono AudRate)
-- Instr (Stereo AudRate)
type InstrMap a = [(InstrumentName, Instr a)]

instrument :: InstrumentName -> Music a -> Music a
tempo :: Dur -> Music a -> Music a -- I *think* 1 = 60 bpm

-- AbsPitch and (AbsPitch, Volume) have ToMusic1 instances
-- Double and (Double, Double) have AudioSample instances
renderSF :: (Clock p, ToMusic1 a, AudioSample b) =>
  Music a ->
  InstrMap (Signal p () b) ->
  (Double, Signal p () b) -- duration and SF we can use to render a WAV
```